(2)

RADC-TR-90-424
Final Technical Report
December 1990

ENGINEERING FOR ARTIFICIAL INTELLIGENCE SOFTWARE

SRI International

John Rushby, Mark E. Stickel, Richard J. Waldinger

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

RADC-TR-90-424 has been reviewed and is approved for publication.

APPROVED: [illegible]

MARK L. FAUSETT
Project Engineer

APPROVED: [illegible]

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: [illegible]

RONALD S. RAPOSO
Directorate of Plans & Programs

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE December 1990	3. REPORT TYPE AND DATES COVERED Final Mar 89 - Jun 90

4. TITLE AND SUBTITLE ENGINEERING FOR ARTIFICIAL INTELLIGENCE SOFTWARE	5. FUNDING NUMBERS C - F30602-87-D-0094 PE - 62702F PR - 5581 TA - QC WU - 05
6. AUTHOR(S) John Rushby, Mark E. Stickel, Richard J. Waldinger	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) SRI International 333 Ravenswood Ave Menlo Park CA 94025	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Air Development Center (COES) Griffiss AFB NY 13441-5700	10. SPONSORING/MONITORING AGENCY REPORT NUMBER RADC-TR-90-424

11. SUPPLEMENTARY NOTES
RADC Project Engineer: Mark L. Fausett, Capt, USAF/COES/(315) 330-7944
(Continued)

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Rule-based systems are being applied to tasks of increasing responsibility. This report focuses on techniques for the verification and validation of these systems. Conventional software-quality assurance depends on the availability of requirements and specification documents. For rule systems, there are generally none because the capabilities of these systems evolve through a development process that is partly experimental in nature. Conventional testing techniques are considered; however, such techniques do not carry over absence of errors. Methods for proving the consistency of rule systems are examined. These methods require that the rules be viewed declaratively, which may be too much of a simplification. A semantics for rule systems based on term rewriting is developed. Standard tests for confluence of term rewriting systems cannot be converted to rule systems, however, because the firing of rules can depend on the absence, as well as the presence, of elements in working memory.

Finally, we consider deductive methods for the validation of rule systems. The system of rules, the operation of the language, and information about the subject domain are represented in a system theory. Validation tasks such as proving termination or verifying properties of the system are phrased as conjectures within the theory. If the conjecture can be proved, the corresponding validation task has been completed.
(Continued)

14. SUBJECT TERMS Rule-based systems, Verification, Validation, Testing, Semantics	15. NUMBER OF PAGES 68
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAR

Block 11 (Continued)

Prepared by SRI under subcontract to IIT Research Institute, Route 26N., Beeches Technical Campus, Rome NY 13440-2069.

Block 13 (Continued)

A method for the gradual formulation of specifications based on the attempted proof of a series of conjectures has been found to be suitable to an evolving rule system. Success on the approach is limited by the power of existing theorem-proving systems. A system SNARK, which is being developed and applied to the proof of validation conjectures for rule systems, is described.

FOREWORD

This is the Final Technical Report, Revision A, CDRL No. G003, for Task 7 under contract F30602-87-D-0094. This contract is with IIT Research Institute (IITRI) and is sponsored by the Rome Air Development Center. The work was performed by SRI International under subcontract R009406 with IITRI providing management support.

Contents

Chapter 1

Introduction

SRI International (SRI) is pleased to present this Final Report, *Engineering for Artificial Intelligence Software*, to Rome Air Development Center, in fulfillment of Contract F30602-87-0-0094. The report describes research on software engineering issues in artificial intelligence (knowledge-based) software.

Our specific focus is on techniques for verification and validation, which are important components of methodologies for conventional software quality assurance. *Verification* is the process of determining whether each level of specification, and the final code itself, fully and exclusively implements the requirements of its superior specification. That is, all specifications and code must be *traceable* to a superior specification. *Validation* is the process by which delivered code is directly shown to satisfy the original user requirements. Verification is usually a manual process that examines descriptions of the software, while validation depends on testing the software in execution. The two processes are complementary: each is effective at detecting errors that the other will miss, and they are therefore usually employed together.

Verification and validation depend on the availability of requirements and specification documents—at least to the extent that it is possible to determine whether a program has experienced a failure. The problem with requirements and specifications for AI software is that generally there are none. The absence of precise requirements and specification documents for much AI software reflects the genuine difficulty in stating à priori the expectations and requirements for a system whose capabilities will evolve through a development process that is partly experimental in nature.

With conventional, sequential programming languages, the behavior of a program in execution has traditionally been understood in terms of the step-by-step execution of its components. More modern treatments strive for a declarative semantics, in which the properties of a program can be deduced by conventional mathematical reasoning, without recourse to a model of procedural execution. Certain programming languages based on logic (for example, OBJ [14] or, in a compromised form, Prolog [7]) possess such declarative semantics.

AI systems are often written with rule-based systems like OPS5 [10,3] or CLIPS [11], which pose extra challenges for verification and validation as compared to conventional programming languages.

Rule-based systems are based on production rules that act on assertions in "working memory." The rules in such a system are often described as a "knowledge-base," rather than as a program, suggesting that they can be understood declaratively—that is without considering interactions among the rules, or the order in which they execute. This is an idealized view; in practice, it *is* necessary to consider interactions among rules and the order of execution. Execution order is determined by the "conflict resolution" strategy of the rule interpreter concerned. (The "instantiations" of rules that are eligible to execute ("fire") in a given cycle form the "conflict set"; the conflict resolution strategy selects one of the rules in this set for firing.) Conflict resolution strategies in practical rule-based systems can be quite complex.

Because conflict resolution strategies have such a profound effect on the behavior of rule-based systems, it is important to explore the extent to which it is possible to accurately model the behavior of these systems in the presence of realistic conflict resolution strategies. It is also important to explore whether it is possible to make statements about the behavior of a rule-based system that will be true for *any* conflict resolution strategy.

In the remainder of this chapter, we give an outline of the report. Chapter 2 discusses issues of testing rule-based systems. Functional or "black-box" testing determines whether a system produces correct results when presented with sample inputs. The concept of "revealing subdomains" for testing only representative elements of equivalence classes of inputs and random generation of test data are key to effective functional testing.

Structural or "white-box" testing identifies execution paths through a program and constructs a set of sample inputs that will cause each path to be executed. The construction of test data on the basis of the structure of the program is likely to yield a more complete test than purely functional testing. However, the notion of execution path is not as well defined for rule-based systems as it is for conventional programming languages. The content of working memory, rather than a program counter, determines which rule to fire next. Criteria and approaches for defining execution paths for rule-based systems are examined.

Testing is often an economical method of finding errors in a system and validating that it satisfies the user's expectations. However, testing is inherently limited: it can demonstrate the presence of errors, but (usually) cannot conclusively demonstrate the absence of errors. The remaining chapters progressively develop concepts and methods designed to provide greater assurance of the correctness of rule-based systems by proving instead of testing properties.

Chapter 3 considers the specific problem of "consistency" in some simple, propositional rule-based systems. A rule-base with a consistency property may give different answers under different execution orders, but it will not give "contradictory" answers. We describe a special-purpose algorithm for testing this notion of consistency for propositional rule-bases with or without negative condition elements.

Chapter 4 presents a term-rewriting-system semantics for rule-based systems. The first-order formalism models systems much more capable than propositional rule-based systems and is realistic for OPS5-like systems. Each production rule is regarded as a rewrite rule in a term rewriting system. Firing a rule corresponds to rewriting a term that represents the contents of working memory.

Modeling rule-based systems as term-rewriting systems allows consideration of widely studied properties of these systems, including the Church-Rosser property, which establishes that the outcome of a computation is independent of the order in which individual steps are executed. A rule-base with the Church-Rosser property will always produce the same final result no matter which eligible rule is selected for firing at each step. Such a rule-base would work the same under *any* conflict resolution strategy. A more interesting question is whether a rule base has a partial Church-Rosser property in the presence of certain conflict resolution strategies. Unfortunately, it is shown that even such a simple conflict resolution strategy as a specificity rule defeats the usual test for the Church-Rosser property, as does the presence of negative condition elements.

Chapter 5 describes our most recent work, which applies very general deductive methods to a variety of verification and validation tasks for rule-based systems. As in Chapter 4, rules are formalized as operations that rewrite the contents of working memory. Typical tasks include showing that a system will satisfy a given requirement, finding conditions in which a particular failure will occur, or establishing that a rule will never be able to fire. For each rule system we construct a corresponding "system theory" and each validation task is carried out by proving that a corresponding conjecture is valid in the system theory. Validation conjectures for sample systems have been proved by SNARK, a new theorem-proving system. Proving a series of validation conjectures can help us formulate a specification for the system. This approach is the most general and powerful of those we have described, since a wide range of properties of rule-based systems can be formalized and proved rather than merely tested.

Chapters 2, 3, and 5 relate to item 1(a)–i of the statement of work concerning construction of specifications for knowledge-based software; Chapters 4 and 5 relate to item 1(a)–ii concerning a semantic characterization of such systems; Chapter 2 relates to items 1(b)–i and 1(b)–ii concerning testing; and Chapters 3, 4, and 5 relate to item 1(b)–iii concerning establishment of a sound theoretical basis for checking rule-based systems. Several of these chapters draw on the work of Judy Crow, of the Computer Science Laboratory.

Chapter 2

Testing of Rule-Based Systems

We follow convention in using the general term "testing" to refer strictly to the notion of *dynamic* testing, in which program behavior is observed as a function of program execution. Conversely, *static* testing refers to analysis of program text, and possibly related formulations such as requirements and specifications, independent of execution behavior. The purpose of dynamic testing is to examine the behavior of the system over a "reasonable" input sample. Given that the input space of most programs is intractably large, a sample is typically defined by partitioning the input space into equivalence classes whose members are expected to exhibit similar behavior. One "representative" from each class is then selected for testing.[1] The equivalence criteria determine which of several dynamic testing strategies is most appropriate. In the following discussion, we focus primarily on two strategies: functional or "black-box" testing and structural or "white-box" testing. We discuss techniques developed for conventional software which also appear productive in the domain of rule-based AI software.

2.1 Functional Testing

The goal of functional testing is to discover discrepancies between the actual behavior of a software system and the desired behavior described in its functional specification. In functional testing, test data are selected with respect

[1]There are of course alternative ways of defining the input sample (*cf.*, for example, [27, pp. 29–30]), but the approach mentioned here appears to be the most widely used.

to a program's *function* as defined by its requirements, specification and design documents—so-called program-independent sources. Several functional testing discussions, including those in [31] and [25], also cite the importance of program-dependent sources, including the code itself. In any case, the relevant sources are used to provide a functional specification which can be viewed as a typically unspecified or only very generally specified relation F on $I \times O$ for input domain I and output domain O. Input and output domains are usually partitioned into groups or classes based on the relevant documents or program-independent/dependent sources; given a certain class of input, a certain class of output results, i.e., $F(i, o)$, for $i \in I, o \in O$. Typically, test data are selected which cover the input and output domains, i.e., input data are chosen which lie well within or just inside/outside the boundaries of each class $i \in I$, and produce output representative of each class $o \in O$.

The general approach of functional testing is directly applicable to rule-based AI software. Of course specific techniques which rely on careful or perhaps even formal specification are less applicable, given the development paradigm for most rule-based software. We have concentrated on a synthesis of two techniques: an adaptation of the "revealing subdomains" method mentioned above [31] and a variation on random testing in the spirit of [15,16].

2.1.1 Revealing Subdomains

As noted in [31], the basic intuition behind the notion of revealing subdomains is quite simple; ***elements of a subdomain behave identically***—either every element produces correct output, or none does. In particular, ***test criterion C is revealing for a subset S*** of the input domain if whenever any element of S is processed incorrectly, then every subset of S which satisfies C fails to execute successfully. Let the predicates OK and $SUCC$ denote successful execution of an element of S and a subset of S, respectively. The formal statement of the preceding intuitive definition is as follows.

$$REVEALING(C, S) \text{ iff}$$
$$(\exists d \in S)(\neg OK(d)) \Rightarrow (\forall T \subseteq S)(C(T) \Rightarrow \neg SUCC(T)) \text{ [31, } p. \text{ 239]}$$

Unfortunately, as Weyuker and Ostrand also note, running successful tests from a revealing subdomain S does not in general guarantee that the program is correct on S; such guarantees are purchased only at a cost equivalent to that of a proof of correctness for the subdomain. On the other hand, we can guarantee that S is revealing for certain specified errors E. A *subdomain* S is *revealing for an error* E if for a program F, such that E is an error in F and E affects some element of S, every element $s \in S$ is affected, i.e., $\neg OK(s)$ [31, p. 239]. Thus, the correct execution of an element from a revealing subdomain guarantees the absence of the specified error on that subdomain. Of course the incorrect execution guarantees only that some (though not necessarily the specified) error has occurred.

Revealing subdomains are constructed by a two-part process as follows. The first step consists of partitioning the input domain into sets of inputs, each of which follows the same or a family of related paths through the program. In conventional software, the partition is based on the program's flow graph. For AI software, either an execution graph or reasonable facsimile will suffice. The second step consists of specifying the *problem partition* and is somewhat less well defined. Weyuker and Ostrand [31, p. 240] state only that partitions should be formed "on the basis of common properties implied by the specifications, algorithm, and data structures." To supplement this somewhat vague directive, we adapt the first three steps of the *category-partition* method for specification-based functional tests developed by Ostrand and Balcer [25, p. 679].[2] Using only program-independent sources, these steps include

1. identify individual functional units which can be separately tested and for each unit identify and characterize parameters and objects in the environment crucial to the unit's function;

2. partition the elements identified in 1 into distinct cases;

3. determine constraints, if any, among the cases identified in 2.

Whatever its precise method of discovery, the purpose of the problem partition is to separate the problem domain into classes which are *in theory* equivalent with respect to the program, whereas the purpose of the path domains is to separate the problem domain into classes which are *in fact* treated identically by the program. Revealing subdomains are defined as the intersection of the two classes, i.e., as equivalence classes of input domain elements which are processed identically by the program and characterized identically by program-independent specifications. By definition, each such

[2]The process enumerated below constitutes only the preliminary analysis suggested by Ostrand and Balcer who describe a method for creating functional test suites using a generator tool to produce test descriptions and scripts.

subdomain has the property that either all or none of its elements are processed correctly. It follows that the actual test data need only consist of an arbitrary element from each subdomain.

2.1.2 Random Generation of Test Data

In a survey of automatic generation of test data, Ince [15] observes that systematic use of randomly generated test data potentially provides reasonable coverage at low cost. The idea, subsequently elaborated in a short note by Ince and Hekmatpour [16], exploits preliminary results independently noted in [9] which indicate that relatively small sets of random test data do appear to provide good coverage. For programs such as AI rule-based software systems which typically have little if any program-independent documentation, random generation of test data seems particularly promising.

2.1.3 A Synthesis

An obvious alternative to either of the techniques mentioned in Sections 2.1.1 and 2.1.2 is their combination. Ideally, this synthesis focuses the low-cost, good-coverage benefits apparently associated with random generation of test data on functionally relevant classes of input identified by the revealing subdomains method. Additionally, the path domains specified by the revealing subdomains method provide a built-in criterion for evaluating the coverage of the randomly generated test data.

2.2 Structural Testing

The goal of structural testing is to expose run-time errors by exercising certain critical execution paths through the program. Execution paths are typically defined with respect to the program's control flow graph; paths are selected on the basis of criteria such as all nodes, all edges, or some combination of nodes and edges. Several researchers have shown that the most effective path selection criteria exploit context, i.e., data- as well as control-flow properties of the program [26,24] and Clarke *et al.* [6] provide a formal evaluation of these and other criteria based on data-flow relationships. While the necessity of both data- and control-flow-based properties appears firmly established, Clarke and her colleagues note that additional studies are needed to consider issues such as the relative cost and detection capabilities of the various path selection criteria.

Unfortunately, the notion of path criteria for rule-based systems is somewhat problematic. There are basically two issues: a productive definition of execution path and, given that, effective path selection criteria, which we discuss in the order given.

2.2.1 A Definition of Execution Path for Rule-Based Software

As noted, the notion of execution path is well defined for conventional software, but decidedly ill-defined for rule-based software. This is the case for several reasons. First, rule bases have both "declarative" and control flow elements; despite the frequent claim that rule bases are strictly declarative, there is often implicit encoding of control information.Thus to the extent that rule bases are declarative, the notion of execution sequence is problematic, and to the extent that control information is implicit, control flow is often difficult to understand and characterize. Second if a rule-based system is considered independently of the associated inference engine, its execution is nondeterministic, further complicating the notion of path.

What, then, is a suitable notion of path for rule bases? There are clearly several desiderata. The notion should be compositional, i.e., it should specify elementary connections between rules and define paths as their transitive closures. Additionally, implicit control flow information should be made explicit. Note that unlike conventional software, where all branches of a predicate or test construct are explicit, rule-based software tends to explicitly represent only the 'successful' branch; rules which are not enabled are effectively ignored.Finally, the notion should focus on relevant execution flow information as opposed to low-level connectivity relationships. The literature includes several proposals for "execution graphs" for rule bases, two of which have been specifically proposed as a basis for structure-based testing, namely the approaches proposed by Stachowitz *et al.* [28] and Kiper [17].

2.2.1.1 Proposals Extant in the AI Literature

Stachowitz and colleagues specify a *Rule Flow Diagram* which is in turn derived from a *Dependency Graph* (DG). A dependency graph is a representation for facts and rules in a knowledge base, where an arc in the graph denotes that a literal in the conclusion (RHS) of rule *a* unifies with a literal in the antecedent (LHS) of rule *b*. Facts are simply rules with empty antecedents. The intuition behind the dependency graph is that an arc connects rules a and b just in case firing rule a can lead to the firing of rule b. For example, there would be an arc from a to b in the DG representation of the following rules.

$$\begin{aligned} a: &\quad A \wedge B \rightarrow X \wedge Z \\ b: &\quad X \wedge Y \rightarrow C \end{aligned}$$

However, there are difficulties with this graph specification. For example, firing rule a above clearly does *not* enable the following rule, despite the fact that rules a and b' satisfy the arc criteria for DGs.

$$b': \; X \wedge \neg Z \rightarrow D$$

A further problem is the apparently unpublished technique for deriving rule flow diagrams from dependency graphs. In rule flow diagrams, nodes represent rules and arcs represent execution sequences. The question is, where does the sequencing information come from? Stachowitz *et al.* appear to suggest that rule flow diagrams can be generated directly from DGs without additional information, but this is surely not the case, as the following example illustrates.[3] The rule set is based on an example in Kiper [17, p. 7].

$$\begin{aligned} 1 &: A \rightarrow B \\ 2 &: B \rightarrow C \\ 3 &: B \wedge C \rightarrow D \\ 4 &: A \rightarrow D \\ 5 &: C \wedge D \rightarrow E \end{aligned}$$

It is difficult to see how a rule flow diagram generated strictly from the DG would reflect the appropriate execution sequence in which rules 2 and 4 jointly enable rule 5. Furthermore, assuming such a procedure exists, it is not clear that it produces a generally satisfactory result; if rule sequencing rather than some notion of causality is the criterion on arcs, information such as the fact that rules 2 and 4 jointly enable rule 5 could be lost.

Finally, and perhaps most important, the DG appears useless for rule-based systems characterized by (re)occurrences of a given set of literals in

[3]Curiously, in the only published test case we could find [4, p. 3], it is not at all clear how the flow diagram is derived from the rule base; no DG is provided and arcs appear from rule 2 to rules 5,6,7,8 despite the fact that there are no literals common to rules 2 and 5-8 in the example given in [4, p. 3].

a large number of rules. In the worst case all rules would be connected; in less extreme cases, however, the problem of excessive connectivity is still significant.

The DG, rule flow diagram pair appears to be the most widely cited of the rule-based analogues to execution graphs, but as suggested above, it is somewhat less than satisfactory. We turn now to the alternative proposed by Kiper.

Kiper [17] suggests a graph construction which explicitly represents the notion of causality. In these graphs, nodes represent rules and arcs denote the relation "enables." Specifically, rule i enables rule j just in case the firing of rule i results in rule j's addition to the agenda. Note that an arc in this type of graph specifically does *not* mean either that as a result of rule i firing, rule j will fire, or that the RHS of rule i unifies with a condition on the LHS of rule j. What it does mean is that the cumulative effect of the chain of rules ending in rule i is to cause rule j to be added to the agenda, and moreover the conditions for j to fire were not satisfied prior to the firing of rule i. In addition, Kiper explicitly represents conjunction and disjunction. Thus Kiper's graph of the preceding five rules would reflect the fact that rules 2 and 4 jointly enable rule 5. More important, Kiper's graph construction is based on a criterion which specifies that the representation for rule bases be independent of any inferencing mechanisms. We think this is a useful criterion. Nevertheless, there appears to be a serious drawback to Kiper's representation: in general, it is not conveniently computable. This follows from the fact that there is no locality condition on arcs; i.e., the existence of an arc from rule i to rule j is a function of the entire path up to and including rule i. For example, consider the two rules below, where the existence of an arc from rule 1 to rule 2 depends on whether the path leading up to rule 1 has already established Y.

$$\begin{aligned} 1: &\quad A \wedge B \rightarrow X \\ 2: &\quad X \wedge Y \rightarrow Z \end{aligned}$$

To summarize, we have analyzed two candidates for graphing the analogue of execution paths for rule-based systems and found both to be deficient with respect to the criteria of compositionality, explicit representation of control flow, and effective representation of information flow proposed at the beginning of this section. In the following section we suggest an alternative notion of execution path for rule-based systems.

2.2.1.2 An Alternative Proposal

The notion of execution path proposed below for rule-based systems reflects execution sequencing and information flow at the level of rule interaction. Note that this differs fundamentally from the notion of control flow typically graphed for conventional software, which reflects sequencing between statements and more fine-grained information-flow, i.e., data-flow properties. For example, control flow graphs for conventional software explicitly represent loop statements, whereas our representation ignores loops and other rule-internal constructs.[4]

An *execution flow graph* for a rule-based system S is a (not necessarily unique) directed graph $G(S) = (N, E, N_i, N_f)$, where N is the (finite) set of nodes, $E \subseteq N \times N$ is the set of edges, and $N_i \subseteq N$, $N_f \subseteq N$ are the sets of initial and final nodes, respectively. Each node in N represents a rule in the rule base of S. For each pair of distinct nodes m and n in N which satisfy constraints C on the rules represented by m and n, there is a single edge (m, n) in E. The constraints, C, are as follows:

1. for every predicate p which appears as the outermost symbol of a term in both the RHS of m and the LHS of n, the two occurrences of p must unify;

2. the LHS of m is consistent with the LHS of n; i.e., the $LHSs$ of the two rules exhibit no overt contradictions.[5]

In this initial formulation, there are no edges of the form (n, n).

An execution flow graph defines the rule-execution sequences or paths within a system S. A *subpath* in $G(S)$ is a finite, possibly empty sequence of nodes $p = (n_1, n_2, \ldots, n_{len(p)})$ such that for all i, $1 \leq i < len(p)$, $(n_i, n_{i+1}) \in E$. We denote the set of all paths in $G(S)$ as $PATHS(S)$. A *cycle* is a subpath of length ≥ 2 which begins and ends at the same node. The graph $G(S)$ is *well-formed* iff every node in N occurs on some path $p \in PATHS(S)$ and $G(S)$ contains no cycles.[6]

[4]Of the three extant AI-based graph representations, only Stachowitz *et al.* graph rule-internal constructs. As noted in Section 2.2.1.1, this granularity has certain drawbacks.

[5]For computational reasons, we don't want to check the consistency of the $LHSs$ of the transitive closure of all rules reachable from m, but it might be productive to set some experimentally determined bound, e.g., check all rules in the length i subpath terminating at m.

[6]The no cycle condition is probably too restrictive, but there is clearly a large class of systemsthat satisfy this constraint.

Let's see how well the proposed graph formalism handles the previous examples, which we reproduce below.

$$
\begin{aligned}
a:&\quad A \wedge B \rightarrow X \wedge Z\\
b':&\quad X \wedge \neg Z \rightarrow D
\end{aligned}
$$

This case is straightforward; although the DG representation erroneously includes an edge from rule a to rule b', the edge is ruled out by constraint 2 in our graph formalism. The next case, derived from an example in Kiper [17], is more challenging. Consider the now familiar rule set below.

$$
\begin{aligned}
1 &: A \rightarrow B\\
2 &: B \rightarrow C\\
3 &: B \wedge C \rightarrow D\\
4 &: A \rightarrow D\\
5 &: C \wedge D \rightarrow E
\end{aligned}
$$

Due to the fact that each rule has a single term RHS, our graph and the DG for this example are identical and appear as shown below.

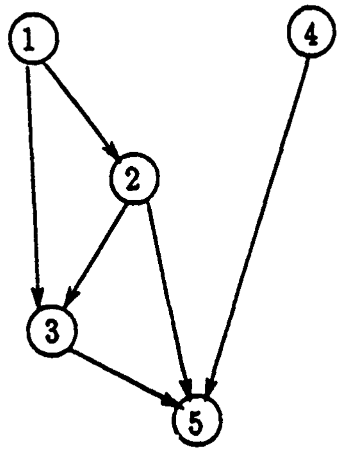

Figure 2.1: Execution Flow Graph for Kiper Rule Set

As given, the graph illustrates three paths which do not correspond to possible execution sequences: [1,2,5; 1,3,5; 4,5]. However, if we postprocess the graph, drawing an arc as shown below between the edges of all nodes which jointly satisfy the *LHS* of their common immediate successor[7], the graph exhibits all and only the correct execution paths for the given rule set.[8]

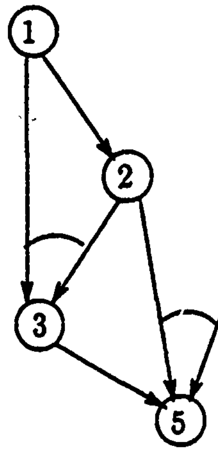

Figure 2.2: Revised Execution Flow Graph for Kiper Rule Set

2.2.2 Path Criteria for Rule-Based Software

As defined in [6], a *path selection criterion* is a predicate which assigns a truth value to any pair (M, P), where M is a program module and P is a subset of *PATHS*(M). Accordingly, a pair (M, P) *satisfies* a criterion C iff $C(M, P) = true$. The purpose of path selection criteria is to identify for testing a *productive* subset of the potentially infinite set of paths through a module, where the notion of productivity is relativized to a particular testing objective. Given the set of well-formed graphs specified by our graph formalism, the set *PATHS*(S) for any rule-based system S is clearly finite. Accordingly, our path selection criterion is modest to the point of vacuity;

[7]Node j is an immediate successor of node i just in case there is an edge from i to j.

[8]We could further stipulate that equivalent paths such as [1,2,3,5] and [1,2+3,5] be "collapsed." It seems likely that the postpass will have to be more sophisticated to handle other less immediate relations between rules. An alternative is to add additional constraints to the constraint set C.

we merely specify complete path coverage, i.e., the equivalent of the *all-paths* criterion defined for conventional software in [26].[9]

In conclusion, many of the testing techniques that have been developed for conventional software do not carry over directly to rule-based systems. Furthermore, while testing can detect errors, it is of limited value in establishing their absence. In the following chapters, we turn to validation methods based on logical proof.

[9]More experience with the graph representation, including a reformulation of the well-formedness condition, may well expose a need for more substantive path selection criteria.

Chapter 3

Proving Consistency of Rule-Based Systems

Several authors have proposed methods for testing rule-based systems for consistency [12,23,29] but none of them present a rigorous definition for rule-base consistency. A promising starting point might be to interpret rules as sentences in a logic and take the interpretation of consistency directly from logic—a theory is consistent if and only if it has a model (i.e., some interpretation of its constant, function, and predicate symbols that valuates all its formulas to true).

It turns out that this approach will not work. In logic, the following set of formulas (where $\longrightarrow$ is interpreted as logical implication) is consistent:

$$\begin{aligned} p &\longrightarrow q \\ p \wedge r &\longrightarrow \neg q \end{aligned}$$

since both formulas evaluate to true in any interpretation that assigns false to p. However, if these formulas are interpreted as rules, and p and r are input variables assigned the value true, then both q and $\neg q$ will be asserted—an obvious contradiction. The problem here is that although the rules are consistent on their own, they form an inconsistent theory when combined with certain initial values. That is, the following "instantiated theory" is inconsistent:

$$\begin{array}{rcl} p & \longrightarrow & q \\ p \wedge r & \longrightarrow & \neg q \\ p & & \\ r & & \end{array}$$

The explanation is, of course, that it is not the rule-base alone that induces the theory corresponding to an expert system—it is the rule-base *plus* the initial facts.

This observation allows us to define ***consistency*** for the theory corresponding to a rule-based system as follows: let I_0 be any interpretation for the inputs to the system (i.e., assignment of initial values),[1] then there must exist a model for the rule-base (interpreted as a theory) which is an extension to I_0.

We can extend this idea to accommodate "user-defined" notions of inconsistency as well. For example, the following set of rules

$$\begin{array}{rcl} p & \longrightarrow & \text{good} \\ p \wedge r & \longrightarrow & \text{bad} \end{array}$$

is not inconsistent in the sense just defined since there is no logical inconsistency in assigning true to both "good" and "bad." If, however, the user considers this assignment to be undesirable, then he may supply the proposition

$$\neg(\text{good} \wedge \text{bad})$$

as a *specification constraint*. The requirement is then to show that any assignment to the inputs of the system can be extended to a model for the rule-base, and any such model will also satisfy the specification constraints.

This notion of consistency leads naturally to an algorithm for testing certain simple (acyclic) propositional rule-bases for the property.

[1]If there is a validity constraint on input values, then we can restrict the interpretations to those which satisfy it.

1. Order the rules so that all those in which a given condition element appears in the RHS precede all those in which it appears in the LHS (i.e., elements must be defined before they are used). If this ordering cannot be satisfied, then the rule base does not have the necessary acyclic property and cannot be tested by this algorithm.

2. Work through each rule in turn, assigning a "label" to each condition element appearing in its RHS as follows:

   - The label for each input element is itself,
   - The "local label" for each element on the RHS is formed by substituting into the LHS the label for each element appearing in the LHS.
   - If a condition element appearing on the RHS already has a label, then update that label by disjoining the local label to it; otherwise assign the local label as the label.

3. When all labels have been assigned, test for logical consistency by showing that the conjunction of the labels for each condition element and its negation is unsatisfiable, and test for specification consistency by showing that any specification constraints, with condition elements instantiated by their labels, are tautological.

A couple of examples should make this clear. If we take the following pair of rules:

$$\begin{aligned} p &\longrightarrow q \\ p \wedge r &\longrightarrow \neg q, \end{aligned}$$

then the inputs are p and r. The first rule yields p as the label for q, while the second yields $p \wedge r$ as the label for $\neg q$. For consistency, we need to test whether the conjunction of the labels for q and $\neg q$—i.e.,

$$p \wedge (p \wedge r)$$

is unsatisfiable. Since it is not (it is satisfied by the assignment of true to both p and r), we conclude that the rule base is inconsistent.

Similarly, for the rules

$$\begin{aligned} p &\longrightarrow \text{good} \\ p \wedge r &\longrightarrow \text{bad} \end{aligned}$$

and the specification constraint

$$\neg(\text{good} \land \text{bad})$$

we compute the label p for "good" and $p \land r$ for "bad" and therefore need to test whether

$$\neg(p \land (p \land r))$$

is a tautology. Since it is not, we conclude that the rule base does not satisfy its specification constraint.

This technique is easily extended to deal with a form of "closed world assumption" in which negative condition elements cannot be asserted directly, but are assumed if the corresponding positive condition element has not been asserted. In this case, the label for a negative condition element is simply the negation of the label for its corresponding positive element. Obviously, in this case only specification constraints need to be checked. The mixed case, in which some negative conditions are asserted directly, and others are assumed from the absence of the positive assertion, can also be accommodated.

The algorithm described here is obviously very limited: it works only for acyclic, propositional rule bases, and ignores conflict resolution. Nonetheless, it accomplishes rather more than Ginsberg's KB-Reducer [12,13], is considerably simpler (and almost certainly faster) and, unlike Ginsberg, we have given a specification for the notion of consistency that is checked by the algorithm. We have implemented our algorithm in Scheme.

Chapter 4

Term-Rewriting Semantics for Rule-Based Systems

We develop a term rewriting system (TRS) semantics for OPS5-like rule-based systems. Our treatment of production systems here differs from earlier presentations. Rule-based systems (also termed production systems) have been defined by Forgy [10,3] as follows.

> A production system is a program composed entirely of conditional statements called *productions.* These productions operate on expressions stored in a global data base called *working memory.* The productions are stored in a separate memory called *production memory.* The production is similar to the If-Then statement of conventional programming languages: a production that contains n conditions C_1 through C_n and m actions A_1 through A_m means
>
> > When working memory is such that C_1 through C_n are true simultaneously, then actions A_1 through A_m should be executed.
>
> The condition part of a production is usually called its *LHS* (left hand side), and the action part is called its *RHS* (right hand side).
>
> The production system interpreter executes a production system by performing a sequence of operations called the *recognize-act cycle*:

1. [Match] Evaluate the LHSs of the productions to determine which are satisfied given the current contents of working memory.
2. [Conflict resolution] Select one production with a satisfied LHS. If no productions have satisfied LHSs, halt the interpreter.
3. [Act] Perform the actions specified in the RHS of the selected production.
4. Go to step 1.

The production system we present here is a simplified version of OPS5. Data in working memory and productions are just atomic formulas instead of terms with attribute-value pairs. Actions, such as input and output, other than those that alter working memory or halt execution, are not present.

Nevertheless, this simplified version captures the essence of OPS5 operation: a recognize-act cycle executes productions to map states described by the contents of working memory to new states described by the new contents of working memory; conflict resolution restricts the application of productions when more than one might be applied.

A semantic characterization of the simplified production system is very useful for understanding the operation of OPS5 and OPS5 programs. It would be easy to extend our system to the exact form of OPS5 terms and their match procedures, but this would not be very fruitful, since the useful and interesting semantic features of OPS5 concern the selection (by conflict resolution) and application of productions.

Likewise, incorporating the semantics of other action types, such as input and output or dynamic addition of productions, would detract from our effort to identify the semantics of the essence of OPS5-like systems. This is consistent with semantic analyses of other systems. For example, the fixpoint semantics of Prolog with negation as failure also ignores input and output and `assert` and `retract` operations [19].

4.1 Elements of OPS5

The data and program of a production system written in an OPS5-like language are stored in *working memory* and *production memory* respectively. The rules in production memory that comprise the program are applied to the data in working memory by the production system interpreter's *recognize-act cycle* that recognizes a rule's applicability and executes its actions. A *conflict resolution strategy* decides which rule to apply if more than one is applicable.

4.1.1 Working Memory

The working memory of an OPS5-like language can be approximately characterized as a set of variable-free atomic formulas.

4.1.2 Production Memory

The production memory of an OPS5-like language can be characterized as a set of rules of the form

$$\textit{conditions} \longrightarrow \textit{actions} \quad ,$$

where the LHS is a set of conditions that must be satisfied for the rule to be applicable and the RHS is a set of actions that are executed when the rule is applied.

Conditions can be expressed by atomic formulas or negated atomic formulas and may contain variables.[1] The atomic formulas in conditions are referred to as *positive* condition elements and the negated atomic formulas are referred to as *negative* condition elements. We require that every variable in a negated atomic formula also appear in an unnegated atomic formula. The rule

$$P_1, \ldots, P_m, \ \textit{not}\ N_1, \ldots, \ \textit{not}\ N_n \longrightarrow \textit{actions}$$

is applicable if there is a substitution σ of variable-free terms for the variables of $P_1, \ldots, P_m$ such that

$$P_1\sigma \in W, \ldots, P_m\sigma \in W, N_1\sigma \notin W, \ldots, N_n\sigma \notin W \quad ,$$

where W is the current contents of working memory.

The most important actions are to *make* a new working memory element, to *remove* a working memory element matched by a condition, to *modify* (the arguments of) a working memory element matched by a condition, or to *halt* execution.

[1]OPS5 working memory elements are atomic formulas. Negated atomic formulas as conditions stipulate absence from working memory of the atomic formula. Some other rule-based systems allow negated atomic formulas as working memory elements [11,30].

4.1.3 Recognize-Act Cycle

The recognize-act cycle finds an applicable rule and executes it. It can be written as

```
until no rule is applicable or
     a halt action has been executed
  select a rule whose LHS is applicable
         to the current contents of working memory
  and execute the actions of its RHS
```

4.1.4 Conflict Resolution Strategy

In the recognize-act cycle, more than one rule may be applicable to the current contents of working memory. In such a situation, the conflict resolution strategy restricts the choice of which rule is to be applied.

The simplest conflict resolution strategy chooses randomly which rule to execute, from the set of applicable rules.

Random choice is often infeasible. More restrictive conflict resolution strategies may be necessary. Some very complex conflict resolution strategies have been developed to fairly tightly constrain choice of rules. Such complexity makes analysis of rule-based systems and specification of their semantics very difficult without specifying the whole conflict resolution strategy in minute detail.

To have some hope of developing analyzable rule-based systems, only relatively simple conflict resolution strategies can be considered. Two criteria often used for rule choice are *recency* of matched working memory elements and *specificity* of condition instantiations. We believe that for many purposes the recency criterion can be ignored, but the specificity criterion must somehow be modeled, since it is such an important aspect of the programming of rule-based systems.

It seems to us that a major purpose of the recency criterion is to yield "focused" behavior of the rule-based system. The recency criterion tends to force the system to apply rules whose conditions match recently added or modified working memory elements. This results in focused system behavior, because the rules selected tend to be those whose inputs (conditions) are the outputs (actions) of the most recently applied rules. This focus makes behavior of the system more comprehensible to the human observer, since the system is more inclined to execute several steps of a single task in sequence, instead of interleaving operations from several different pending tasks. We choose to ignore recency as a criterion that must be modeled, since alternative execution orders may still be intended to yield the same final result.

On the other hand, modeling the specificity criterion seems to be critical. The specificity criterion is used to program special vs. general case behavior. If $LHS_1 \longrightarrow RHS_1$ and $LHS_2 \longrightarrow RHS_2$ are two rules such that LHS_1 is more specific than LHS_2 (i.e., there is a substitution θ such that $LHS_1\theta \subseteq LHS_2$), then the first rule, if it applies, will be executed in preference to the second.

The more specific rule can be used either to refine or to displace the behavior of the general rule. If it applies, the more specific rule is executed. We assume that the rule's actions alter working memory so that the rule is no longer applicable.[2] The conditions of the less specific rule may remain satisfied after the more specific rule has been executed, or some of its conditions may no longer be satisfied by working memory. In the former case, both rules are executed, and the results of the general-case rule are added to those of the special-case rule; in the latter case, only the special-case rule results are computed.

4.2 Term Rewriting Systems

A term rewriting system is a set of rules

$$LHS \longrightarrow RHS$$

where the variables of RHS are all variables of LHS. A rule $LHS \longrightarrow RHS$ of a term rewriting system can be applied to a term t if some subterm u of t is an instance $LHS\sigma$ of LHS. In that case, $t(u)$ can be rewritten to $t(RHS\sigma)$, i.e., the subterm u that matched LHS is replaced by the corresponding instance of RHS.

[2]A "refractoriness" criterion in the conflict resolution strategy can be used to allow the conditions of the rule to remain satisfied, but inhibit the rule's repeated execution. We will ignore modeling such criteria in this analysis.

Term rewriting systems can be used to perform equational reasoning, where $LHS \longrightarrow RHS$ is a directed (because left-hand sides are replaced by right-hand sides and not vice versa) version of the equality $LHS = RHS$. For example, the set of rules

$$\begin{array}{rcl}
0+x & \longrightarrow & x \\
x+0 & \longrightarrow & x \\
x+(-x) & \longrightarrow & 0 \\
(-x)+x & \longrightarrow & 0 \\
-0 & \longrightarrow & 0 \\
-(-x) & \longrightarrow & x \\
(x+y)+z & \longrightarrow & x+(y+z) \\
-(x+y) & \longrightarrow & (-y)+(-x) \\
x+((-x)+y) & \longrightarrow & y \\
(-x)+(x+y) & \longrightarrow & y
\end{array}$$

is a set of rules for some equalities of group theory with addition function $+$, inverse function $-$, and identity element 0.

Term rewriting systems have been extensively studied and there are many interesting properties that can be explored.

The most notable properties that a term rewriting system may have are *termination* and *confluence*. A term rewriting system has the termination property if no term t can be rewritten as an infinite sequence of terms, i.e., for no t, $t \longrightarrow t_1 \cdots \longrightarrow t_i \cdots$. Like the halting problem for Turing machines, determining whether a term rewriting system has the termination property is undecidable in general, though it can often be decided in specific cases. The set of rewriting rules above has the termination property.

A term rewriting system has the confluence property if for any term t that can be rewritten in two ways: $t \longrightarrow \cdots \longrightarrow t'$ and $t \longrightarrow \cdots \longrightarrow t''$ there is a term s such that $t' \longrightarrow \cdots \longrightarrow s$ and $t'' \longrightarrow \cdots \longrightarrow s$. In effect, the confluence property states that regardless of which rewriting rule is applied whenever more than one is applicable, one can still reach the same result.

Term rewriting systems that are both terminating and confluent are called complete. They have the very desirable property that if t_1 and t_2 are equal in the equality theory of the rules, then the irreducible term t_1^* that results from rewriting t_1 until no rule is applicable is identical to the irreducible term t_2^* that results from rewriting t_2 until no rule is applicable. The set of rewriting rules above is a terminating and confluent and thus complete set of rewriting rules for the theory of free groups. For term rewriting systems with the termination property, it is decidable to determine if the system is confluent. Moreover, the Knuth-Bendix method [18] that is used as the decision procedure for confluence sometimes succeeds in extending nonconfluent term rewriting systems to confluent ones. For example, the complete term rewriting system above can be automatically derived from the axioms of a free group:

$$\begin{aligned} 0+x &= x \\ (-x)+x &= 0 \\ (x+y)+z &= x+(y+z) \quad . \end{aligned}$$

A further possible property of term rewriting systems that is relevant to our effort to define a term-rewriting-system semantics for OPS5-like languages (since working memory is variable-free (i.e., *ground*)) is *ground confluence.* A term rewriting system may be confluent on all ground terms, even if it is not confluent on all terms, which may include variables. Unfortunately, determining ground confluence is undecidable in general.

4.3 A Rewriting Interpretation of OPS5

Just as the rules of a term rewriting system rewrite a term, the rules of a rule-based system can be viewed as rewriting the contents of working memory to the new contents of working memory. This viewpoint allows OPS5 rules to be reformulated to omit reference to the procedural notions of making, removing, and modifying working memory elements.

Consider the rule

$$A, B, C, \ not\ D \longrightarrow modify(A), remove(B), make(E) \quad .$$

This can be reformulated as a rewriting rule in which the RHS specifies which atomic formulas replace the working memory elements that match positive condition elements A, B, C:

$$A, B, C, \ not\ D \longrightarrow A', C, E \quad .$$

Formally, let P_k be the set of atomic formulas that appear in positive condition elements in the LHS of production k and N_k be the set of atomic formulas that appear in negative condition elements in the LHS of production k.

The applicability of production k to working memory W with substitution θ can be defined by

$$applicable(k, W, \theta) \equiv (P_k\theta \subseteq W) \land (W \cap N_k\theta = \emptyset) \quad .$$

Let A_k be the set of atomic formulas that appear in the RHS of production k.

Suppose production k is applicable to working memory W with substitution θ. Then the result of executing production k on working memory W with substitution θ is W', where

$$W' = (W - P_k\theta) \cup A_k\theta \quad .$$

OPS5 programs are defined to halt execution if no production is applicable or a halt action is executed. The latter condition can be reduced to the former by a transformation on the set of rules: Create an atomic formula named *halt* and add ***not halt*** as a condition element to the LHS of each production; include the ***halt*** atomic formula in the RHS of each reformulated production whose RHS included a halt action. For example, the rules

$$\begin{array}{ccl} LHS_1 & \longrightarrow & RHS_1 \\ & \vdots & \\ LHS_i & \longrightarrow & RHS_i, halt \\ & \vdots & \\ LHS_n & \longrightarrow & RHS_n \end{array}$$

is reformulated as

$$\begin{array}{ccl} LHS_1, \textit{ not halt} & \longrightarrow & RHS_1^* \\ & \vdots & \\ LHS_i, \textit{ not halt} & \longrightarrow & RHS_i^*, halt \\ & \vdots & \\ LHS_n, \textit{ not halt} & \longrightarrow & RHS_n^* \quad . \end{array}$$

The RHS element *halt* in the original set of rules refers to the halt action; in the reformulated rules, the element *halt* that appears negated in the conditions and in the RHS is the atomic formula *halt*, whose presence in working memory may be created by rule *i*, and whose absence is required for the applicability of every rule.

4.4 Analysis

Viewing OPS5-like systems as term rewriting systems permits a less procedural, more abstract and logical expression of programs. The working memory, now expressed in presence and absence conditions in the LHS and replacement formulas in the RHS, can be regarded as a set of terms which can be reasoned about.

The term-rewriting-system viewpoint allows us to ask questions about OPS5-like systems that parallel those about term rewriting systems, e.g., questions of termination and confluence. With termination assumed, confluence is a desirable property that assures that the same conclusion will be derived regardless of the choice (suitably restricted by the conflict resolution strategy) of which rule to execute at each point. Even if the system is deliberately nonconfluent, it would be desirable to learn something of the extent and nature of the system's indeterminacy by testing for confluence. Unfortunately, complete confluence tests for conditional and priority term rewriting systems, which the transformed OPS5-like systems resemble, do not exist [22].

Efforts to extend standard Knuth-Bendix confluence tests to transformed OPS5-like systems have failed so far and demonstrate that negative condition elements and conflict resolution by specificity both pose difficulties for determining confluence.

For example, consider the set of rules

$$\begin{aligned} A &\longrightarrow B \\ A &\longrightarrow C \\ B &\longrightarrow D \\ C, \mathit{not}\ E &\longrightarrow D \quad , \end{aligned}$$

which contains a negative condition element *not E*. The standard Knuth-Bendix confluence test proves the confluence property for ordinary term rewriting systems by demonstrating local confluence: any time two rules with overlapping LHSs are both applicable, the results of the two rule applications can both be reduced to the same final result. The only overlap in this example is between $A \longrightarrow B$ and $A \longrightarrow C$ and results B and C can be reduced to the same final result D. However, A, E reduces to B, E and C, E, which can be reduced to final results D, E and C, E, so the rules are not confluent. The problem is that the counterexample to confluence A, E is not the result of overlapping a pair of rules. Exhaustive generation of inputs or exhaustive symbolic execution can discover such instances of nonconfluence, but is likely to be costly and incomplete.

Also consider the rules

$$\begin{array}{rcl} A & \longrightarrow & B \\ A & \longrightarrow & C \\ B & \longrightarrow & D \\ C & \longrightarrow & D \\ C, E & \longrightarrow & F \quad , \end{array}$$

with the rule with LHS C, E taking precedence by specificity over the rule with LHS C. Like the previous set of rules, this set is confluent on input A, but not A, E.

That negative condition elements and conflict resolution by specificity should yield similar difficulties for confluence testing is not surprising, since sets of rules ordered by specificity may be translatable into sets of rules that do not require conflict resolution by specificity by adding negative condition elements to the more general case rules. For example, translating the rules $C \longrightarrow D$ and $C, E \longrightarrow F$ into C, *not* $E \longrightarrow D$ and $C, E \longrightarrow F$ eliminates the need for conflict resolution by specificity.

Simple confluence tests for OPS5-like systems, and determining other formal properties easily, will probably require further simplification of the systems, such as eliminating negative condition elements and conflict resolution strategies—perhaps unacceptable changes to the systems—or more global, exhaustive analysis that may take account of the finite universe of formulas that may occur in working memory.

Chapter 5

Proving Properties of Rule-Based Systems

5.1 Introduction

Languages based on rules are an appealing implementation vehicle for expert systems. The system ean be developed incrementally without much preliminary planning. In introducing a new rule, one supposedly need have little understanding of how the rest of the system behaves. The rules may embody the advice of many different experts, who are ignorant of each other's opinions and may even disagree with each other. Proponents of rule-based methodologies have found that they can develop running systems far more quickly than with conventional programming languages.

As a consequence of this success, expert systems based on rules have been proposed for tasks of increasing responsibility, including aircraft and spacecraft fault diagnosis as well as financial and medical advice. For this reason, the question arises of how we can establish that these systems will be worthy of our confidence?

This is where a conflict emerges. Accumulated experience suggests that to be trustworthy, a system must be constructed in a systematic way that begins with an attempt to formulate its specification prior to the implementation effort. This doctrine is antithetical to the rule-based system methodology, in which the intended behavior of the system changes at each stage of its implementation. The system is developed in the absence of specifications; in fact, the methodology may be regarded as a framework for rapid prototyping, in which we gradually formulate an executable specification through experimentation. On the other hand, a complex nondeterministic system of rules is rarely acceptable as a specification; it is difficult for anyone, including its developers, to predict what it will do.

It is not the purpose of this chapter to criticize or improve the rule-based system methodology. Rather, we shall attempt to apply deductive techniques to support the methodology as it is practiced. We shall provide techniques to determine what a rule system does, to identify its faults, and to establish confidence in it. One may be able to formulate a single specification that characterizes the intended behavior of the system. That specification may then be used as the basis for a reimplementation of the system using conventional software-engineering techniques. We may hope that the reimplemented system will be more efficient, concise, and reliable than the original.

In other cases, in which the system is too complex to allow a full specification to be verified or even formulated, we can use deductive methods to assist in the testing of the system. We can generate sets of test cases that exercise all the rules of the system or that cause certain inconsistencies or anomalies to occur. We can detect that certain rules will never be executed. The same deductive framework can serve a variety of these purposes.

Because rules look like logical sentences, it is tempting to treat them that way, and analyze them for properties such as consistency. In fact, rules cannot usually be understood in a purely declarative way. They are imperative constructs with the intended side effects of adding and deleting elements from a single data structure, the "working memory." In this report, we treat a rule language as an imperative language. Because all side effects in the language alter a single structure, the language is more amenable to logical analysis than most imperative languages, such as those with general assignment statements.

Special problems arise because of the nondeterministic nature of rule-based languages. A situation can occur in which more than one rule is applicable, and the system implementation must choose between them. Different implementations may make different choices and a system may behave correctly in one implementation and not in another. It may be difficult to anticipate what different implementations may do.

Conventional program verification often assumes that a full specification for a correct system is available. In this work, we recognize that the system may be incorrect and the specification may be only partial. We detect faults and formulate the specification gradually, as a result of attempts to prove a series of conjectures about the system. A further option is to attempt to prove the conjecture's negation, which will hold if the system fails to possess the desired property.

In most work on program verification, a proof gives us at best a yes/no answer as to whether a system meets its specification. Implicit within a proof, however, is other potentially valuable information, which is usually discarded. For example, if we prove the existence of a fault in a system, we may be able to extract from the proof a description of the conditions under which that fault occurs. Program synthesis techniques for extracting programs from constructive proofs (e.g., Manna and Waldinger [20]) may also be applied to extract other sorts of information.

5.1.1 Validation Tasks

In keeping with these goals, we consider a variety of validation tasks, most of which have both a positive and a negative aspect.

- (+) Verification: Proving that a system will always satisfy a given condition.

  (−) Fault detection: Exhibiting an input that causes a system to fail to satisfy a given condition.

- (+) Termination: Proving that a system will always terminate.

  (−) Loop detection: Exhibiting an input that will cause a system to fail to terminate.

- (+) Firing: Exhibiting an input that causes a given rule to fire.

  (−) Redundancy: Proving that no input will cause a given rule to fire.

- (+) Consistency: Proving that no input can produce an inconsistent working memory.

  (−) Inconsistency: Exhibiting an input that will produce an inconsistent working memory.

Some of these problems are significantly more difficult than others. For example, to prove that a system always terminates will generally require considering all execution histories beginning from any possible input, at least in principle. Exhibiting an input that causes Rule A to fire may require considering a small number of inputs and only part of the system. Thus, we can expect to be successful at this smaller task more readily than at establishing termination.

5.1.2 The System Theory

Our approach is deductive. For a given system of rules, we develop a *system theory*, which is defined by a set of axioms that express the actual behavior of the system. The system theory also incorporates any background knowledge we wish to take into account in validating the system. For each validation task, there is an associated conjecture. If we can manage to establish the validity of the conjecture in the system theory, we have performed the associated validation task. Typically, to perform the negative aspect of a task, we prove the negation of the conjecture associated with its positive aspect. If we want to exhibit an input or other object as part of our task, we must restrict the proof to be sufficiently constructive to tell us how to build such an object. The description of the object can then be extracted from the proof.

5.1.3 Related Work

There have been several efforts to apply conventional testing techniques to rule-based systems (e.g., Becker et al. [1], Kiper [17]). The body of work closest to ours is that of Chang, Combs, and Stachowitz [28,4,5] at the Lockheed Artificial Intelligence Center. The Lockheed work, like ours, deals with some specifications of the expected properties of the rule-based system and uses deductive methods to establish them. It uses Prolog as an inference system, which limits it to properties expressed as Horn clauses. The SNARK theorem prover we are developing accepts properties in a full first-order logic, with equality and mathematical induction. Our work is also original in that it presents a unified theoretical framework for expressing and establishing properties of rule-based systems.

5.1.4 Outline of this Chapter

We first provide a description of a somewhat idealized rule language in Section 5.2. For a given system of rules, we show how to construct a corresponding system theory in Section 5.3 and in Section 5.4 how to translate validation tasks into conjectures in the theory. In Section 5.5, we exhibit portions of a validation proof and present a scenario leading to the formulation of a specification for a textbook rule-based system. Finally, in Section 5.6 we describe the SNARK theorem prover we have been developing to prove validation conjectures and to extract information from validation proofs.

5.2 Rule-Based Systems

In this section we present a prototype rule-based system framework that will serve as the focus of our effort.

5.2.1 The Rule Language

Our rule language is a smoothed-up version of OPS5 [10,3]. A rule describes an operation to be performed on working memory. The *working memory* is a (finite) set of atoms $p(t_1, \ldots, t_k)$, where p is a predicate symbol and $t_1, \ldots, t_k$ $(k \geq 0)$ are terms. The atoms in working memory are *ground*, that is, they contain no variables, only constant, function, and predicate symbols. An atom $p()$ with no arguments will be written p. For example,

$$\{farmer(john), banker(mother(john))\}$$

is a working memory.

A *rule* is an expression of the form

$$L_1, \ldots, L_m \longrightarrow R_1, \ldots, R_n \quad .$$

Here each element L_i of the left side is a *literal*, that is, either an atom $p(t_1, \ldots, t_k)$ or the negation *not* $p(t_1, \ldots, t_k)$ of an atom. Each element R_i of the right side is an (unnegated) atom. We do not require rule elements to be ground, that is, they may contain variables. For example

$$red(x),\ not\ big(x) \longrightarrow blue(x)$$

is a rule. We impose certain restrictions on rules; these will be discussed after we have described rule application.

5.2.2 Rule Application

To apply a rule to working memory, we first select a ground instance of the rule, that is, we replace each of its variables with a corresponding ground term. We require that the instances of the positive (unnegated) atoms on the left side of the rule be present in working memory and that the instances of the negated atoms be absent.

For example, the rule

$$Rule\ B:\ red(x),\ not\ big(x) \longrightarrow blue(x)$$

is applicable to the working memory

$$\{red(b), big(a)\}.$$

The appropriate rule instance is obtained by replacing x with the ground term b. The instance $red(b)$ of the positive atom $red(x)$ is present in this working memory; the instance $big(b)$ of the negated atom $big(x)$ is absent.

To apply an applicable rule to the working memory, we delete the selected instances of the positive atoms of the left side and add the instances of the atoms of the right side. For example, to apply Rule B to the working memory $\{red(b), big(a)\}$, we delete $red(b)$ and add $blue(b)$, to obtain the new working memory

$$\{blue(b), big(a)\}.$$

Note that instances of the positive atoms on the left side are deleted as the rule is applied. This means that if any of these atoms is to be retained, it must appear on the right side as well, so that it can be added back into working memory.

For example, the rule

$$red(x), big(x) \longrightarrow blue(x)$$

will delete instances of both $red(x)$ and $big(x)$ from working memory. If it is intended that the rule retain the instance of $big(x)$, the rule must read

$$red(x), big(x) \longrightarrow blue(x), big(x).$$

(This rule describes a situation in which big red blocks are to be painted blue, but remain big.) The rationale for this convention is that the positive atoms of the left side are replaced in working memory by the atoms of the right side; thus the rule behaves as a rewriting of working memory.

One restriction we impose on the language is that every variable that occurs anywhere in a rule must occur in some positive atom on the left side. This implies that once we have instantiated these positive atoms, we have instantiated the entire rule. For example, the rule

$$red(x),\ not\ big(y) \longrightarrow red(z)$$

violates this restriction for two reasons: the variable y from the negated atom $big(y)$ and the variable z from the right side are not present in the positive atom $red(x)$ on the left side. If we attempt to apply this rule, it is unclear which instance of $big(y)$ is to be absent from working memory. It is also unclear which instance of $red(z)$ is to be added.

We do not allow explicit negation signs in the working memory. This is not an essential limitation. If it is desired to express that an atom $p(t_1, \ldots, t_k)$ is false, we can introduce a new predicate symbol $negp$, and include $negp(t_1, \ldots, t_k)$ in the working memory, with the understanding, to be expressed in the theory, that $negp(t_1, \ldots, t_k)$ is the complement of $p(t_1, \ldots, t_k)$.

A *rule system* is an (unordered) set of rules. To apply a rule system to a working memory, we repeatedly apply any of the rules to the working memory until no rule is applicable. The final working memory is the result of applying the system. Application of a rule system is nondeterministic; by selecting different rules, or different instances of the same rule, we may obtain different results.

5.2.3 Explicit Halting

Our systems halt only when no rule is applicable. Some rule-based languages offer an explicit *halt* statement: if the special symbol $HALT$ appears on the right side of a rule that is applied, the system will halt at once, even if some rule is applicable. This is a convenience that does not increase the logical power of the language. As we mentioned in Chapter 4, any system with the halt feature can be transformed into one that behaves the same way without the feature. The transformed system includes the negated atom *not halt* (that is, *not halt*()) on the left side of each rule. If any rule contains the special symbol $HALT$ on its right side, it is transformed into a rule with the atom *halt* on its right side instead. If such a rule should fire, it adds the atom *halt* to working memory. Then no rule will be applicable, and the transformed system will halt, without invoking any special halt feature.

For example, the system

$$\begin{array}{l} red(x) \longrightarrow blue(x) \\ yellow(x) \longrightarrow HALT \end{array}$$

with the halt feature is transformed into the system

$$\begin{array}{l} red(x),\ not\ halt \longrightarrow blue(x) \\ yellow(x),\ not\ halt \longrightarrow halt \end{array}$$

without the halt feature. The two systems behave the same way.

5.2.4 Conflict Resolution Strategy

When several rules are applicable to the same working memory, the system invokes a conflict resolution strategy to choose a single rule to be applied. Conflict resolution strategies tend to be complex. The spirit of the rule-based methodology suggests that the correctness of the system should not depend on the details of the strategy. The choices of the strategy may influence the efficiency of the system or the understandability of its execution sequence, but the correctness of the final outcome should be independent of these choices. Consequently, we have developed an approach that will perform our validation tasks regardless of the conflict resolution strategy. If some aspects of the strategy turn out to be crucial to the correctness of the system, they may be expressed in an augmented system theory.

An exception is made in the case of the specificity aspect of the conflict resolution strategy, which does affect the correctness of the system. According to the specificity principle, a more specific rule is to be preferred to a more general one. This principle allows us to state a rule in its greatest generality, and then to add exceptions by introducing new rules, without the need to qualify the original rule. For example, in the system

$$\begin{array}{ll}
\textit{Rule } 1: & bird(x) \longrightarrow bird(x), fly(x) \\
\textit{Rule } 2: & bird(x), penguin(x) \longrightarrow bird(x), penguin(x), negfly(x) \\
\textit{Rule } 3: & bird(x), penguin(x), inairplane(x) \longrightarrow \\
 & bird(x), penguin(x), inairplane(x), fly(x)
\end{array}$$

each rule is more specific than the previous one, which it qualifies. It might be erroneous to apply the earlier rule if the later rule were applicable. For example, the second rule,

$$bird(x), penguin(x) \longrightarrow bird(x), penguin(x), negfly(x)$$

should not be applied if the penguin is in an airplane, because then the third rule should supersede it. Thus the second rule has the implicit condition *not* $inairplane(x)$.

Because specificity has a bearing on correctness concerns, we do want it reflected in the system theory. We achieve this by transforming rules so that the implicit conditions imposed by the strategy are made explicit. For example, Rule 2 would be transformed to

$$\begin{array}{ll}
\textit{Rule } 2': & bird(x).penguin(x), \textit{ not } inairplane(x) \longrightarrow \\
 & bird(x), penguin(x), negfly(x)
\end{array}$$

Rule 2′ cannot be applied if Rule 3 is applicable. Rule 1 above would be transformed into

$$\textit{Rule } 1': \ \textit{bird}(x),\ \textit{not penguin}(x),\ \textit{not inairplane}(x) \longrightarrow \textit{bird}(x), \textit{fly}(x)$$

Rule 1′ cannot be applied if either Rule 2′ or Rule 3 is applicable.

We shall apply this transformation before forming the system theory, so that the specificity aspect of the conflict resolution strategy will be reflected in our validation.

5.3 The System Theory

In this section, we describe how a given rule system is described in a corresponding system theory, so that questions about the system may be phrased as conjectures within the theory.

Consider a rule

$$P_1, \ldots, P_i,\ \textit{not } Q_1, \ldots,\ \textit{not } Q_j \longrightarrow R_1, \ldots, R_l$$

with variables $x_1, \ldots, x_k$.

We define a relation $\textit{applic}(r, \langle t_1, \ldots, t_k \rangle, w)$, which holds if rule r is applicable to working memory w with variables $x_1, \ldots, x_k$ instantiated to ground terms $t_1, \ldots, t_k$, respectively, by the axiom

$$\textit{applic}(r, \langle t_1, \ldots, t_k \rangle, w) \equiv \begin{bmatrix} P_1[t_1, \ldots, t_k] \in w \\ \wedge \\ \vdots \\ \wedge \\ P_i[t_1, \ldots, t_k] \in w \\ \wedge \\ Q_1[t_1, \ldots, t_k] \notin w \\ \wedge \\ \vdots \\ \wedge \\ Q_j[t_1, \ldots, t_k] \notin w \end{bmatrix}$$

for all ground terms $t_1, \ldots, t_k$ and any working memory w. Here $\langle t_1, \ldots, t_k \rangle$ is a tuple of ground terms and r is a constant that names the rule. We write $P[t_1, \ldots, t_k]$ for the result of replacing each variable $x_1, \ldots, x_k$ in P

with the corresponding term $t_1, \ldots, t_k$. In other words, the appropriate instances of the positive atoms must be present in working memory and the corresponding instances of the negated atoms must be absent. A separate such axiom is provided for each rule of the system.

For example, for the rule

$$\textit{Rule } 1: \textit{parent}(x,y),\ \textit{not male}(x) \longrightarrow \textit{parent}(x,y), \textit{mother}(x,y)$$

we provide the axiom

$$\textit{applic}(\textit{rule}1, \langle s,t\rangle, w) \equiv [\textit{parent}(s,t) \in w \wedge \textit{male}(s) \notin w].$$

Note that each predicate symbol in a rule is represented by a function symbol in the system theory. Thus, *parent* and *male* are function symbols.

We also define a function $\textit{apply}(r, \langle t_1, \ldots, t_k\rangle, w)$, whose value is the result of applying the rule r to working memory w with variables $x_1, \ldots, x_k$ instantiated to ground terms $t_1, \ldots, t_k$, by the axiom

$$\begin{array}{lllll}
\textit{if} & \textit{applic}(r, \langle t_1, \ldots, t_k\rangle, w) & & & \\
\textit{then} & \textit{apply}(r, \langle t_1, \ldots, t_k\rangle, w) = w & - & P_1[t_1, \ldots, t_k] \\
 & & & \vdots \\
 & & - & P_i[t_1, \ldots, t_k] \\
 & & + & R_1[t_1, \ldots, t_k] \\
 & & & \vdots \\
 & & + & R_l[t_1, \ldots, t_k].
\end{array}$$

Here $w - u$ and $w + u$ are the results of deleting the element u from the set w and adding u to w, respectively.

For example, for Rule 1, we provide the axiom

$$\begin{array}{llll}
\textit{if} & \textit{applic}(\textit{rule}1, \langle s,t\rangle, w) & & \\
\textit{then} & \textit{apply}(\textit{rule}1, \langle s,t\rangle, w) = w & - & \textit{parent}(s,t) \\
 & & + & \textit{parent}(s,t) \\
 & & + & \textit{mother}(s,t).
\end{array}$$

When we leave a variable free, we mean it to have implicit universal quantification. Also, we use our vocabulary to indicate the sorts of the objects involved. For example, the above axiom is to apply to any working memory w and all ground terms s and t.

Note that when a positive atom occurs on both the left and the right sides of the rule, the corresponding term is first deleted from and then added to the set w in the *apply* axiom. Because the term is certain to occur in the set, this will always leave the set the same. For example, in the *apply* axiom for Rule 1, the term $parent(s,t)$ is first deleted, then added. To simplify the axioms (and the corresponding proofs), these operations will be omitted. The *apply* axiom for Rule 1 will actually read

$$if\ applic(rule1, \langle s,t\rangle, w)\ then\ apply(rule1, \langle s,t\rangle, w) = w + mother(t).$$

This translation of rules into axioms depends on our formulation of the rule language; the translation mechanism for other languages will differ slightly.

The following axiom tells us that if a rule is applicable to a working memory, that rule must be one of the rules of the system:

$$\begin{array}{rl} if & applic(r,t,w) \\ then & r = rule1 \vee \cdots \vee r = rulen, \end{array}$$

where t is a tuple of ground terms.

More complex versions of this axiom may be substituted if one desires to express more subtle aspects of the conflict resolution strategy.

We shall say that the entire system is applicable to a working memory w, denoted by $appl(w)$, if some rule, with some instantiation, is applicable. This is expressed by the axiom

$$appl(w) \equiv (\exists r,t) applic(r,t,w).$$

A working memory w to which no rule is applicable, that is, $\neg appl(w)$, is called a *final* working memory.

5.3.1 Histories

A history is a description of a finite initial segment of a possible computation of the system. In the theory, a *history* is a finite tuple of pairs

$$\langle\langle r_1,t_1\rangle,\ldots,\langle r_n,t_n\rangle\rangle.$$

Each r_i is the rule applied at the i^{th} stage of the computation. Each t_i is a tuple of ground terms indicating how the variables of the rule r_i are instantiated at the i^{th} stage.

A history $h = \langle\langle r_1,t_1\rangle,\ldots,\langle r_n,t_n\rangle\rangle$ is *applicable* to a working memory w, denoted by $hist(h,w)$, if there is a finite sequence of working memories

$$w_1, w_2, \ldots, w_{n+1},$$

where $w = w_1$, such that

$$w_{i+1} = apply(r_i, t_i, w_i);$$

that is, each memory is obtained from the previous one by applying the corresponding rule from the history. This is expressed by the axioms

$$hist(\langle\rangle, w)$$
$$hist(\langle r, t\rangle \bullet h, w) \equiv applic(r, t, w) \wedge hist(h, apply(r, t, w))$$

for all working memories w, rules r, tuples of ground terms t, and histories h. Here $\langle r, t\rangle \bullet h$ is the history obtained by inserting the pair $\langle r, t\rangle$ at the beginning of the history h.

The result $sys(h, w)$ of applying an applicable history h to a given working memory w is expressed by the axioms

$$sys(\langle\rangle, w) = w$$
$$\textit{if } hist(\langle r, t\rangle \bullet h, w) \textit{ then } sys(\langle r, t\rangle \bullet h, w) = sys(h, apply(r, t, w)).$$

The second axiom states that the result of applying a nonempty history is the same as that of applying the first rule and instantiation, and then applying the remainder of the history. The result is itself a working memory.

Note that a history describes an initial segment of a computation of a rule system, not necessarily a full computation; more rules may be applicable to the resulting working memory. We say that h is a terminating history starting from w, denoted by $ter(h, w)$, if h is applicable to w and results in a final working memory. This is expressed by the axioms

$$ter(\langle\rangle, w) \equiv \neg appl(w)$$
$$ter(\langle r, t\rangle \bullet h, w) \equiv applic(r, t, w) \wedge ter(h, apply(r, t, w)).$$

It can be established that

$$ter(h, w) \equiv hist(h, w) \wedge \neg appl(sys(h, w)).$$

5.3.2 Finite Sets and Unique Names

Because we use finite sets of expressions to represent working memory, it is necessary to incorporate the theory of finite sets into our system theory. In particular, we include axioms that describe the set addition function $w + v$

and the set membership relation $u \in w$:

$$u \notin \{\}$$
$$u \in w + u$$
$$if\ u \in w\ then\ u \in w + v$$
$$if\ u \neq v\ then\ if\ u \in w + v\ then\ u \in w.$$

For the set deletion function $w - v$ we have

$$v \notin w - v$$
$$if\ u \in w - v\ then\ u \in w.$$

We include a general well-founded induction principle, described in the next subsection, which applies to finite sets and finite histories as well.

Properties of tuples, for reasoning about histories and about tuples of ground terms, are also included but will not be presented here. Theories of finite sets and tuples are described more fully in Manna and Waldinger [21].

Although it may seem pedantic, we must include axioms that tell us that distinct function symbols correspond to distinct predicate symbols in working memory. For example, *red*(s) and *blue*(t) cannot stand for the same atom. This is expressed by the axiom

$$red(s) \neq blue(t).$$

We must provide such an axiom for each pair of function symbols corresponding to predicate symbols in working memory.

We must also provide an axiom stating that if two terms are distinct, they cannot be made identical by applying any of the predicate symbols from the working memory; e.g., for the predicate symbol *red*, we have

$$if\ red(t_1) = red(t_2)\ then\ t_1 = t_2.$$

A similar axiom is provided for each function symbol corresponding to a predicate symbol from working memory.

5.3.3 Well-Founded Induction

Many proofs require use of an induction principle. We use well-founded induction (also called Noetherian induction). This principle has the following form.

To prove a sentence

$$(\forall w)P[w]$$

prove the *inductive step*

$$(\forall w)\left[\begin{array}{l} \textit{if } (\forall w')\,[\textit{if } w' \prec w \textit{ then } P[w']] \\ \textit{then } P[w] \end{array}\right].$$

Here $\prec$ is a well-founded relation, that is, one that admits no infinite decreasing sequences

$$w_1 \succ w_2 \succ w_3 \succ \cdots$$

We provide definitions of many known well-founded relations. For example, the proper subset relation is known to be well-founded over the finite sets because there are no infinite sequences of finite sets

$$w_1 \supset w_2 \supset w_3 \supset \cdots$$

We also include as lemmas other properties of the proper subset relation, e.g.,

$$\textit{if } u \in w \textit{ then } w - u \subset w.$$

Well-founded relations over the tuples are also provided.

5.4 Validation Conjectures

Many validation tasks may be phrased as conjectures within the system theory; if we can establish the validity of the conjecture in the theory, we have carried out the validation task.

For example, suppose we wish to determine whether, for the given rule system, big, red objects will be painted yellow. We may phrase this task as:

$$(\forall w, h, t)\left[\begin{array}{ll} \textit{if} & \textit{red}(t) \in w \wedge \\ & \textit{big}(t) \in w \wedge \\ & \textit{ter}(h, w) \\ \textit{then} & \textit{yellow}(t) \in \textit{sys}(h, w) \end{array}\right].$$

If we can prove this sentence in the system theory, we have shown that the system satisfies the condition.

Note that the sentence does not establish termination of the system, but only that if a history does terminate, the condition will be satisfied. To show that the system does always terminate, it suffices to establish the following *termination condition*:

$$\textit{if } \textit{applic}(r, t, w) \textit{ then } \textit{apply}(r, t, w) \prec w$$

for some well-founded relation $\prec$. In other words, we show that each applicable rule reduces the size of working memory with respect to some well-founded relation. Because well-founded relations do not admit infinite decreasing sequences, this means we must ultimately reach a working memory to which no rule is applicable, i.e., a final working memory.

In our examples, we require the user to provide a well-founded relation for the termination proof. For example, the user might define

$$w_1 \prec w_2 \equiv \begin{bmatrix} (\forall t)\,[\mathit{if}\ red(t) \in w_1\ \mathit{then}\ red(t) \in w_2] \\ \wedge \\ (\exists t')\,[red(t') \in w_2 \wedge red(t') \notin w_1] \end{bmatrix}.$$

In other words, with respect to $\prec$, one working memory is less than another if it has fewer red objects. This is well-founded because working memories are finite.. (We could define this relationship in terms of the number of red objects in working memory, but this would require reasoning about nonnegative integers, as well as sets, in the proof.) A more ambitious effort, which we have not yet attempted, would require the system to discover the well-founded relation as part of the proof process.

If we fail to prove that all red, big objects will be painted yellow, we may attempt to establish the opposite, namely, that some red, big objects will not be painted yellow. This can be established by proving the negation of the original conjecture, i.e., that

$$(\exists w, h, t) \begin{bmatrix} red(t) \in w\ \wedge \\ big(t) \in w\ \wedge \\ ter(h, w) \wedge \\ yellow(t) \notin sys(h, w) \end{bmatrix}.$$

Proving this conjecture will establish the falseness of the original condition. If we restrict the proof to be sufficiently constructive, we may extract a description of a case in which the condition fails to hold. In particular, we obtain a description of an initial working memory, a terminating history, and an object such that the object is initially big and red, but executing the history will produce a final working memory in which the object is not painted yellow.

Suppose we cannot prove termination and we suspect that our system sometimes fails to terminate. To prove this, we must provide a description of a possible infinite execution history, in the form of three functions, r, t, and w, which compute the i^{th} rule, instantiation, and working memory, respectively. These functions must satisfy the property

$$\begin{array}{l} applic(r(i), t(i), w(i)) \wedge \\ apply(r(i), t(i), w(i)) = w(i+1) \end{array}$$

for all integers $i \geq 1$. We have not yet experimented with proving nontermination.

If we want to establish that a particular rule, say Rule 1, can fire, we may prove the conjecture

$$(\exists w, h, t)[hist(h, w) \wedge applic(rule1, t, sys(h, w))].$$

If we restrict the proof to be sufficiently constructive, we may obtain descriptions of the initial working memory w, history h, and instantiation t, such that executing h in initial working memory w will produce a working memory in which Rule 1 is applicable, with instantiation t. Proving the negation of the above conjecture will establish that Rule 1 is redundant, i.e., cannot be executed under any circumstances.

The notion of consistency depends on an application domain. Our rule language does not include explicit negation, but it may include predicate symbols that are understood to be mutually inconsistent. We may expect, for example, that an object cannot be both red and blue, or that the predicate symbol *negp* is to be the complement of the predicate symbol *p*.

If we want to show that our system can never produce an object that is simultaneously red and blue, we may attempt to prove

$$(\forall w, h, t)\left[\begin{array}{ll} \textit{if} & hist(h, w) \\ \textit{then} & \neg[red(t) \in sys(h, w) \wedge blue(t) \in sys(h, w)] \end{array}\right].$$

The above conjecture implies that an object cannot be both red and blue at any stage of the computation. If we are only concerned with the final state of the computation, we may prove

$$(\forall w, h, t)\left[\begin{array}{ll} \textit{if} & ter(h, w) \\ \textit{then} & \neg[red(t) \in sys(h, w) \wedge blue(t) \in sys(h, w)] \end{array}\right].$$

If we can prove the negation of either of the above conjectures, we have established that the system can produce an inconsistency, in either an intermediate state or a final state, respectively. Restricting the proofs to be sufficiently constructive will enable us to extract a description of how the inconsistency can occur.

5.5 Example: The Billing Category System

To illustrate the formation of a system theory, we adapt an example from a standard expert-systems text (Brownston et. al. [3]). The system is to assign each of a fixed, finite pool of customers to a billing category, either normal or priority, depending on the history of the customer. The rules of the system are as follows:

$$\textit{Rule } 1: \ \ good(x),\ not\ set(x) \longrightarrow good(x), set(x), priority(x).$$

That is, if the category of a good customer has not been set, assign the customer to the priority category.

$$\textit{Rule } 2: \ \ bad(x),\ not\ set(x) \longrightarrow bad(x), set(x), normal(x).$$

That is, if the category of a bad customer has not been set, assign the customer to the normal category.

$$\textit{Rule } 3: bad(x),\ not\ set(x), long(x) \longrightarrow bad(x), set(x), long(x), priority(x).$$

That is, if the category of a bad but long-term customer has not been set, assign the customer to the priority category. Note that, by the specificity principle, Rule 3 is intended to supersede Rule 2 when both are applicable; that is, Rule 2 is not meant to apply to long-term customers.

5.5.1 The Rule Axioms

These rules are represented by the following axioms in the system theory.

$$if\ applic(r,t,w)\ then\ r = rule1 \vee r = rule2 \vee r = rule3.$$

In other words, the only rules that can be applicable to a given customer t are Rule 1, Rule 2, or Rule 3.

Axioms for Rule 1:

$$applic(rule1, t, w) \equiv good(t) \in w \wedge set(t) \notin w$$

Note that because the rules have only one variable, we simplify the axioms by using t, rather than $\langle t \rangle$, throughout.

$$if\ applic(rule1, t, w)\ then\ apply(rule1, t, w) = w + set(t) + priority(t)$$

Because the atom $good(x)$ occurs on both sides of Rule 1, the corresponding term $good(t)$ is neither deleted nor added by the axiom.

Axioms for Rule 2:

$$applic(rule2, t, w) \equiv bad(t) \in w \;\wedge long(t) \notin w \;\wedge set(t) \notin w$$

Note that we include in the applicability axiom for Rule 2 the condition, implied by the specificity principle, that the rule should not be applied to long-term customers.

$$if\ applic(rule2, t, w)\ then\ apply(rule2, t, w) = w + set(t) + normal(t)$$

Axioms for Rule 3:

$$applic(rule3, t, w) \equiv bad(t) \in w \wedge long(t) \in w \wedge set(t) \notin w$$

$$if\ applic(rule3, t, w)\ then\ apply(rule3, t, w) = w + set(t) + priority(t)$$

The other axioms of the system theory are the same from one system to the next.

5.5.2 Conjectures: A Scenario

Suppose we wish to determine whether a good customer will always be placed in the priority category. Then we may conjecture

$$(\forall w, h, t)\left[\ if\ ter(h, w)\ then\ \left[\begin{array}{ll} if & good(t) \in w \\ then & priority(t) \in sys(h, w) \end{array}\right]\right].$$

In fact, we cannot prove the above conjecture. We can, however, prove its negation

$$(\exists w, h, t)\left[\begin{array}{l} ter(h, w) \wedge \\ good(t) \in w \wedge \\ priority(t) \notin sys(h, w) \end{array}\right].$$

If we restrict the proof to be constructive, we may extract a description of the initial working memory,

$$w : \{good(t), set(t)\}$$

and the history

$$h : \langle\rangle,$$

the empty history. (The variable t is not instantiated during the proof, so it can be replaced by any ground term that denotes a customer.) In other words, no rule is applicable to the initial working memory $\{good(t), set(t)\}$, because customer t has been marked as if his billing category has already been set. Therefore, the final working memory $sys(h, w)$ is the same as the initial working memory w, and $priority(t) \notin w$.

We attempt to refine our conjecture accordingly. We speculate that if a good customer has no set billing category, he will ultimately be placed in the priority category. We attempt to prove

$$(\forall w, h, t)\begin{bmatrix} if & ter(h, w) \\ then & if\ good(t) \in w \wedge set(t) \notin w \\ & then\ priority(t) \in sys(h, w) \end{bmatrix}.$$

Again we fail to prove this, but succeed in proving its negation. If the proof is restricted to be constructive, we may extract the (inconsistent) initial working memory

$$w : \{bad(t), good(t)\}$$

and the one-element history

$$h : \langle\langle rule2, t\rangle\rangle.$$

This example has again defied our expectations. Because customer t is bad as well as good (and not a long-term customer), Rule 2 can be applied, producing the final working memory

$$sys(h, w) : \{bad(t), good(t), set(t), normal(t)\}.$$

In other words, good customer t has not been put into the priority category.

This scenario illustrates the pitfalls we may face in formulating a specification for a rule-based system (or any system). It also illustrates how a proof system may help us break through some preconceptions. With further experimentation, we may attempt to formulate and prove a full specification for a rule system, which characterizes its intended behavior. For the billing category system, we propose the following conjecture:

$$if\ (\forall t)\begin{bmatrix} [good(t) \in w \underline{\vee} bad(t) \in w] \wedge \\ set(t) \notin w \wedge \\ priority(t) \notin w \wedge \\ normal(t) \notin w \end{bmatrix}$$

$$\textit{then}\ (\forall h)\left[\begin{array}{ll} \textit{if} & ter(h,w) \\ \textit{then} & [priority(t) \in sys(h,w) \equiv good(t) \in w \vee long(t) \in w] \wedge \\ & [normal(t) \in sys(h,w) \equiv bad(t) \in w \wedge long(t) \notin w] \wedge \\ & set(t) \in sys(h,w) \wedge \\ & [good(t) \in sys(h,w) \equiv good(t) \in w] \wedge \\ & [bad(t) \in sys(h,w) \equiv bad(t) \in w] \wedge \\ & [long(t) \in sys(h,w) \equiv long(t) \in w] \end{array}\right]$$

for all customers t and working memories w. In other words, we suppose that initially each customer is either good or bad, but not both ($\underline{\vee}$ is exclusive *or*). Initially no customer has had his billing category set, and no customer is in either normal or priority category. Then for each terminating history, we require that the customers in the final priority category be those that are initially good customers or long-term customers. Customers in the final normal category must be those that are initially bad customers and not long-term customers. Every customer must have his final billing category set. Furthermore, we may expect that customers in the final good-customer, bad-customer and long-term customer categories are the same as those that were in those categories initially.

The above conjecture relates the initial and final working memories. For the proof to succeed, we must generalize the theorem to relate the intermediate and final working memories. The generalized conjecture implies the above conjecture as a special case. We have not yet proved the above conjecture.

Termination of the system must be proved separately, by establishing the usual termination condition

$$\left[\begin{array}{ll} \textit{if} & applic(r,t,w) \\ \textit{then} & apply(r,t,w) \prec w \end{array}\right]$$

for some well-founded relation $\prec$. In this case, the well-founded relation $\prec$ may be defined by

$$w_1 \prec w_2 \equiv \left[\begin{array}{c} (\forall t)\,[\textit{if}\ set(t) \in w_2\ \textit{then}\ set(t) \in w_1] \\ \wedge \\ (\exists t')\,[set(t') \in w_1 \wedge set(t') \notin w_2] \end{array}\right].$$

In other words, with respect to $\prec$, one working memory is less than another if it has more customers with set billing category. This is well-founded because there are only a finite number of customers.

5.6 The SNARK System

To prove validation conjectures, a theorem prover requires an unusual combination of features. In particular, it must

- Prove sentences in full first-order logic.
- Deal expeditiously with equality and ordering relations.
- Prove theorems by mathematical induction.
- Handle finite sets and tuples.
- Restrict proofs to be sufficiently constructive to allow information extraction when necessary.
- Prove simple theorems without human assistance.

While some existing theorem provers excel in certain of these areas, they are typically deficient in others. The Argonne system [32], for example, is proficient at full first-order logic with equality, but has no facilities for proof by induction. The Boyer-Moore theorem prover [2] specializes in proof by induction, but does not allow full first-order quantification. The Nuprl system [8] is certainly expressive enough—it allows full quantification and proof by induction—but is not geared to finding proofs automatically. Furthermore, it relies entirely on a constructive logic that may prove cumbersome when no information needs to be extracted from the proof.

For these reasons, we have been developing a new theorem prover, SNARK, for application in software engineering and artificial intelligence. SNARK is especially appropriate for the validation of rule-based systems.

SNARK operates fully automatically and uses an agenda to order inference operations. Similarly to the Argonne system, SNARK attempts to compute the deductive closure of a set of formulas. The user selects the inference operations and starting formulas to be used. Agenda elements are formulas in the set of support to be operated upon by all selected inference rules, and are ordered by symbol count.

The most important inference operations available in the system are binary resolution and paramodulation. These rules allow SNARK to deal with predicate logic with equality, which underlies the system theory for rule-based systems. We have used extended versions of these inference rules that are applicable to nonclausal formulas as well as clauses. Hyperresolution can be simulated by control restrictions on the use of binary resolvents. Both clausal and nonclausal subsumption are available to eliminate redundant formulas.

Formulas can be simplified by user-given or derived equalities or equivalences. Innermost or outermost simplification strategies can be specified. Derived equalities can be oriented automatically by Knuth-Bendix or recursive decomposition simplification orderings. Truth-functional simplification is accomplished by rewriting rules, which makes it easy to add new connectives and their simplification rules.

SNARK can use either nonclausal formulas or the more restrictive clauses. If clauses are to be used, SNARK can automatically translate more general formulas to clauses. Even if clauses are primarily used, translation of formulas to clauses is not required to be done only at the beginning of the proof. Rewrites can specify that an atomic subformula of a formula be rewritten to a formula; the result of rewriting may be a nonclausal formula that is later simplified to clause form. For example, the rewrite

$$u \in w + v \equiv u = v \vee u \in w$$

would result in the clause $a \notin s{+}x \vee C$ being rewritten to $\neg(a = x \vee a \in s) \vee C$, which could be replaced by two clauses.

Efficient formula- and term-indexing methods—a choice of path indexing or discrimination-tree indexing—are used to efficiently retrieve the relevant formulas or terms for inference, subsumption, and simplification operations. Efficient indexing is essential for solving difficult problems that require derivation of a large number of results.

SNARK uses sorted logic to efficiently represent the information that certain classes of objects being manipulated are disjoint. In the examples of this chapter, several sorts are used: rules, working memories, working memory elements, customers, history lists, and the pairs that are history list members.

SNARK supports the use of special unification (and subsumption and equality) algorithms. Associative-commutative subsumption is widely used for truth-functional simplification, and commutative matching is used to efficiently implement symmetry of the equality relation.

Although at an early stage of development, SNARK has already been useful in proving properties of rule-based systems, including those indicated in the scenario (Section 5.5.2).

5.7 Summary and Plans

Our work suggests that deductive methods are appropriate to support testing and other validation tasks, besides verification, for rule-based systems. Preliminary results in applying the new deduction system SNARK to sample rule-based systems have been promising. By restricting the system to be sufficiently constructive, we have been able to extract information other than simple yes/no answers from proofs. We have found that a method for formulating specifications by proposing a series of conjectures is appropriate to rule-based systems.

We intend to develop the system theory to apply to more realistic rule languages and to extend SNARK to more complex rule systems and more sophisticated properties and conjectures. The extension will be carried out by introducing inference rules targeted to the application (e.g., rules for reasoning about sets and ordering relations), strategic deduction (e.g., special treatment for inductive proofs), interactive controls, and parallel search for proofs.

Bibliography

[1] Lee Becker, Peter Green, R. J. Duckworth, Jayant Bhatnagar, and Adam Pease. Evidence flow graphs for VV&T. In *Preliminary Proceedings IJCAI-89 Workshop on Verification, Validation and Testing of Knowledge-Based Systems*, 1989. Detroit, MI.

[2] R.S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[3] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, Reading, MA, 1985.

[4] Chin Chang and Rolf Stachowitz. Testing expert systems. In *Proceedings of the Space Operations Automation and Robotics (SOAR-88) Workshop*, 1988. Dayton, OH.

[5] Chin Chang, Rolf Stachowitz, and J.B. Combs. Testing integrated knowledge-based systems. In *IEEE International Workshop on Tools for AI*, 1989. Fairfax, Virginia.

[6] Lori Clarke, Andy Podgurski, Debra Richardson, and Steven Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, November 1989.

[7] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* Springer-Verlag, New York, NY, 1981.

[8] R. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Englewood Cliffs, NJ, 1986.

[9] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10(4):438–443, April 1984.

[10] Charles L. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, July 1981.

[11] Joseph C. Giarratano. *CLIPS User's Guide.* Artificial Intelligence Center, Lyndon B. Johnson Space Center, June 1988. Version 4.2 of CLIPS.

[12] Allen Ginsberg. A new approach to checking knowledge bases for inconsistency and redundancy. In *Proceedings, Third Annual Expert Systems in Government Symposium*, pages 102–111, Washington, DC, October 1987. IEEE Computer Society.

[13] Allen Ginsberg. Knowledge-base reduction: A new approach to checking knowledge-bases for inconsistency and redundancy. In *Proceedings, AAAI 88 (Volume 2)*, pages 585–589, Saint Paul, MN, August 1988.

[14] Joseph A. Goguen and Timothy Winkler. Introducing OBJ. Technical Report SRI-CSL-88-9, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1988.

[15] D.C. Ince. The automatic generation of test data. *Computer Journal*, 30(1):63–69, February 1987.

[16] D.C. Ince and S. Hekmatpour. An empirical evaluation of random testing. *Computer Journal*, 29(4):380, August 1986.

[17] James Kiper. Structural testing of rule-based expert systems. In *Preliminary Proceedings IJCAI-89 Workshop on Verification, Validation and Testing of Knowledge-Based Systems*, 1989. Detroit, MI.

[18] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–293. Pergamon, New York, NY, 1970.

[19] J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, West Germany, 1984.

[20] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2:90–121, 1980.

[21] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming*, volume 1: Deductive Reasoning. Addison-Wesley, 1985.

[22] C.K. Mohan. Priority rewriting: semantics, confluence, and conditionals. In *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, pages 278–291, Chapel Hill, NC, 1989.

[23] Tin A. Nguyen, Walton A. Perkins, Thomas J. Laffey, and Deanne Pecora. Knowledge base verification. *AI Magazine*, 8(2):65–79, Summer 1987.

[24] Simeon Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, November 1984.

[25] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[26] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[27] John Rushby. Quality measures and assurance for AI software. Contractor report 4187, NASA, October 1988.

[28] Rolf Stachowitz, Jacqueline Combs, and Chin Chang. Validation of knowledge-based systems. In *Proceedings of the Second AIAA/NASA/USAF Symposium on Automation, Robotics and Advanced Computing for the National Space Program*, 1987. Arlington, VA.

[29] Motoi Suwa, A. Carlisle Scott, and Edward H. Shortliffe. An approach to verifying completeness and consistency in a rule-based expert system. *AI Magazine*, 3(4):16–21, Fall 1982.

[30] S.A. Vere. Relational production systems. *Artificial Intelligence*, 8(1):47–68, February 1977.

[31] Elaine Weyuker and Thomas Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, SE-6(3):236–246, May 1980.

[32] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning*. Prentice Hall, Englewood Cliffs, NJ, 1984.